N O T I C E

THIS DOCUMENT HAS BEEN REPRODUCED FROM MICROFICHE. ALTHOUGH IT IS RECOGNIZED THAT CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED IN THE INTEREST OF MAKING AVAILABLE AS MUCH INFORMATION AS POSSIBLE

JPL PUBLICATION 80-16

(NASA-CR-163258) AN APPLICATION OF SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE (Jet Propulsion Lab.) 41 p HC A03/MF A01 CSCL 09B

N80-26057

Unclas
G3/61 23618

An Application of Software Design and Documentation Language

E. D. Callender
T. B. Clarkson
C. E. Frasier

June 1980

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL PUBLICATION 80-16

An Application of Software Design and Documentation Language

E. D. Callender
T. B. Clarkson
C. E. Frasier

June 1980

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under NASA Contract No. NAS7-100.

PREFACE

This paper was presented at the AIAA 2nd Computers In Aerospace Conference, October 23-25, 1979. This version is expanded from that which was presented at the conference.

ABSTRACT

This paper discusses the application of SDDL to the detailed software design of the Command Data Subsystem for the Galileo spacecraft. A set of constructs was developed and applied. The paper contains an evaluation of these constructs and examples of their application.

CONTENTS

Figures

Tables

PRECEDING PAGE BLANK NOT FILMED

SECTION 1

INTRODUCTION

One of the major problems that face the software designer is how to record and maintain the detailed design of a piece of software during development. It is desirable that when the design is completed that the associated documentation that records the design also be completed. This report presents a possible solution to this problem. The focus is on the detailed design of the software for the Command and Data Subsystem (CDS) of the spacecraft for Project Galileo. The tool used to support the approach is Software Design and Documentation Language (SDDL).[1] SDDL is a software program design language and associated processor. There are five basic concepts described in this report. They are:

(1) Detailed design can be made a manageable task by restricting its scope. It is neither functional, architectural or general design, nor is it implementation where the clerical details necessary to translate the detailed design into code are added. Detailed design starts once all the major components have been identified and ends when the cookbook of instructions has been created that would allow a clerk to emulate the functions of a computer and execute the program. In more exact terms, a detailed design is an exact and unambiguous description of: the interrelationships (interfaces) between independent logical elements; logical disposition and organization of the data; and interaction of an abstract machine with its data base (algorithms).

(2) There are eight dimensions necessary to define a design. Reference is made to Table 3-1 containing the eight items necessary and, we hope, sufficient to describe a detailed design.

(3) The best realization of a detailed design is in the document that describes that design. This of course assumes that code has not yet been created.

(4) An isomorphism exists between data and control structures.

(5) Five different types of structures (modules) are sufficient to describe a design. The five types are data, control, task, requirement structures, and free form text blocks.

This report is divided into five sections. The first section contains a very brief description of SDDL. The next is a description of the detailed design procedure followed by a description of the SDDL syntax used to support that detailed design. Then the Command Data Subsystem (CDS) is briefly described. The final section relates the experience using SDDL on CDS.

---

[1] Henry Kleine, Software Design and Documentation Language, JPL Publication 77-24, Revision 1, Jet Propulsion Laboratory, Pasadena, California, August 1, 1979.

SECTION 2

DESCRIPTION OF SDDL

The Software Design and Documentation Language (SDDL), contrary to the implication in its title, is a general-purpose processor to support a language for the description of any system, structure, concept, or procedure that may be presented from the viewpoint of a collection of hierarchical entities linked together by means of binary connections. The language comprises a set of rules of syntax, primitive construct classes (module, block, and module invocation), and language control directives. The result is a language with a fixed grammar, variable alphabet and punctuation, and an extendable vocabulary. SDDL represents a rather primitive, but powerful capability. It assists the software designer by providing structure. It supports the naming of pieces of the software design, called modules, the illustration of substructure within a module by automatic indentation, the identification of other module invocation with any module and the creation of a module hierarchy and cross reference list. It operates against a single input file of textual material that is the current description of a design of a piece of software and creates a single report that contains the above information.

SECTION 3

DETAILED DESIGN PROCEDURE

The point of departure for this report is at the start of the detailed design for a piece of software. It is assumed that the reader is familiar with the software design process. A top-level hierarchical program structure has already been created during the general design process.

This document does not address the issue of how to create a detailed design for a piece of software. However, certain basic steps in that process are assumed. It is assumed that for each module, a completed detailed design will be prepared before that module is coded. The level of detail in the design is such that one SDDL statement should on the average generate three to five lines of code, assuming that a higher order language is used or that macros are used in the case of assembly language. It is also assumed that the implementation language is procedural.

The creation of the detailed design for software can be viewed as the creation of a number of structures (MODULEs) and the associated ties between these structures (in SDDL parlance, MODULE INVOCATIONs). There are four very important classes of structures that are created as the detailed design for the software is prepared. They are the structure or list of requirements, the top-level task structure, the hierarchical structure for the programs (control processes), and the associated data structures.

A software package or design is an abstract item. The only physical manifestation of a piece of software is that of its associated documentation. This documentation can exist in many different forms. One form is code. Another form is a detailed design. The outline given in Table 3-1 gives the items that are required for a task, process or data module. It is the kernel outline for the documentation of any module, whether the module be a high, intermediate or low level. The order in the program structure in which these items are generated is dependent upon the design methodology employed by the individual designer and the individual designer's experience and insights into the particular problem at hand. The SDDL processor is insensitive to the order in which these items are generated and processed. What is described below are the constructs that will assist the designer as he/she treats the problems of input data, local data structures, output data and the myriad of other details necessary to complete the detailed design for a particular software package.

The hierarchical program structure has the program as the top entry in the structure. A program may be made up of one or more procedures, subroutines and/or functions. Each subroutine or function may consist of one or more procedures. Modules, as described in SDDL, are sized with a view to facilitate the understanding of the reader. The size of the module when it is described by the designer may be substantially different than the size of a module when it is compiled or assembled into executable code. For example, a subroutine when compiled may consist of its top level module plus a large number of submodules. To this end, individual program modules should be scaled such that only one operation or function is performed by that procedure

and that the implementation will not require an excessive amount of code or structure. Typically this means that the number of design statements (execution directives, data declarations, etc.) required to describe a procedure is ten to fifty; the number of associated lines of code is less than one hundred when the implementation language is either a higher order langauge or a macro assembler language and that the design of a single module will not require more than six or seven levels of structure. This hierarchical program structure only partially describes the flow of control within the program. As the hierarchical structure for the program is being created the control flow for each module must be established. The control flow is described using standard structured programming constructs. (Such structured programming constructs are described in SDD. using the BLOCK construct family.)

Table 3-1. Module Outline

	DATA MODULE (NAME)	PROCESS MODULE (NAME)	TASK MODULE (NAME)
1.0	Module Abstract	Module Abstract	Module Abstract
2.0	Allocated Requirements	Allocated Requirements	Allocated Requirements
3.0	Design Description/ Purpose/Functional Description	Design Description/ Purpose/Functional Description	Design Description/ Purpose/Functional Description
4.0	Affected Module Modifying Process Modules Accessing Process Modules Invoking Data Modules Invoked Data Modules	Affected Modules Invoking Process Modules Invoked Process Modules Accessed Data Modules Modified Data Modules	Affected Modules Invoking Tasks Invoked Tasks Invoked Process Owned Data
5.0	Operating Environment	Operating Environment	Operating Environment
6.0	System Parameters	System Parameters	System Parameters
7.0	Data Specification Access Algorithm/ Protocol Assumptions/ Constraints Statistics	Process Specification Mode of Execution Assumptions/Constraints	Task Specification
8.0	Data Organization Normal Data Error Data	Control Flow Normal Processing Error Processing	Task Organization

In a similar manner, a hierarchical data structure is organized into groups, files, queues, stacks and tables. The data constructs are concerned with the abstract design, not the physical implementation. Each data construct has an associated abstract access method. After the detailed design is completed and before the associated code is generated, it is necessary to determine the physical disposition and representation of the abstract data structures that have been created.

The detailed design of a piece of software is completed when the data structures and control structures have been sufficiently specified that the computer functions could be simulated by hand without making any further design decisions. Using this criterion, coding is the conversion of a detailed design (targeted for execution within a human mind) into the syntax acceptable for input into an assembler or compiler.

One aspect of the method described in this paper is to make a clear distinction between the activities associated with the creation of the detailed abstract software design and the physical implementation of that design. This distinction is at times very subtle. However, the distinction is extremely important and must be kept in mind at all times as the detailed design is being prepared. Care must be taken to ensure that the abstract design is easily mapped into the physical design. The requirement to achieve this has influenced the particular choice of SDDL constructs.

Between the completion of the detailed design and the generation of associated code, some implementation decisions must be made. They can include the selection of the coding language and hardware and the determination of the physical disposition and representation of the appropriate data structures. Of course, decisions on coding language and hardware are many times made at the beginning of the project by default, politics, or other nontechnical reasons. The intent of this design specification approach is to support the capability to postpone decisions of an implementation nature until the detailed design is complete.

The use of SDDL as a computer software design medium exploits the analogous relationship between the hierarchical representation of the data portion of a software system and the program portion. Indeed, the execution of a program may be viewed as the construction of a file of data (instructions) passed to the ALU, where the branches, together with the states upon which they are based, are understood to be the selection characteristics of that file. Both are organized hierarchically by means of extended constructs in the families: MODULE, BLOCK, and MODULE INVOCATION. The SDDL language provides the construct families as primitives together with the capability to add or delete members of each construct family. The language does not provide explicit definitions for the use (semantics) of these constructs. The selection of the abstract constructs presented in this report was made based upon the experience of the authors in designing software. Hence, in many cases the "usual" implementation realization of the abstract construct may be inferred from the similarity to the English words. However, it is the responsibility of the designer to ensure that for any particular design, each of the logical constructs that he uses can be realized in the physical implementation. For examples of these constructs, the reader is referred to the material in Section 4 of this report.

The constructs in the MODULE family allow the user to set up generic classes of modules including data modules such as FILE, STACK, QUEUE, and TABLE, control process modules such as PROGRAM, SUBROUTINE, FUNCTION, and PROCEDURE, and requirement modules. The constructs in the BLOCK family allow the user to create a hierarchical structure within a particular module. If the module is a control process structure, the user will recognize the hierarchical structure as a typical indented set of IF, SELECT, LOOP and PERFORM. The constructs in the MODULE INVOCATION family allow the user to create the network that relates the various modules. Here constructs such as CALL, DO, OPEN, CLOSE, READ, WRITE, and CLEAR are used. MODULE INVOCATION can be thought of as a one way mapping from the interior of the module under discussion to some other module, viewed atomically. For example, the construct CALL can be thought of as a mapping from a subroutine to another subroutine.

The actual steps used in the detailed design process for the CDS are given below. The interfaces and data structures were first designed and then the program control flow was added. There was a substantial amount of iteration within all of the steps.

(1) Define and characterize interfaces between the software to be produced and the environment in which the software will operate. This environment includes interfaces to other software and any hardware interfaces that are used.

(2) Define data structures to be used by the software.

(3) Decompose software into procedures and identify all procedure calls.

(4) Specify control flow.

(5) Identify error conditions and recovery processing.

(6) Verify design, iterate.

SECTION 4

SDDL SYNTAX TO SUPPORT DETAILED DESIGN PROCEDURE

There are five fundamental types of constructs used in the description of a software design. They are: requirements, data, program, task, and unstructured text. Associated with each of these module types are various substructures and invocation rules. A requirements module is a device that allows the designer to state in a structured manner the requirements that the design must satisfy.

Data is described from two points of view: the structure of the data, and the attributes or values that the data may take. The SDDL constructs presented in this section are directed toward a description of the various structures that are used in describing data organization. Because of the very large number of data types that are possible, there are no SDDL constructs presented in this manual directed toward describing data attributes. However, in many of the examples that are given below, attributes are presented as textual information. Hence, a description of a collection of data items in SDDL should contain both structure and attribute information. For example, the scope associated with a data structure or the range of values that a data item may attain is attribute information and is best captured as in-line textual comments in the data structure.

A program (a process/control structure) is described in terms of its type, the process steps that are performed when the program is invoked and the other processes that this program invokes. The SDDL constructs are based upon standard structured programming constructs. As in the case of data representation, no attempt has been made to catalogue the myriad of operators and algorithms that the designer may select to describe the operations performed within a design.

A task is a construct that is introduced to allow a designer to express real-time or interrupt driven designs in SDDL. A task is an independent abstract machine. A task has associated with it data that it owns (machine state) and the executable programs (definition of the function of the machine) that it uses.

The final construct class is concerned with the documentation of the design. Constructs are employed which allow the designer to record all information about the design in SDDL. The intent is to allow the designer to record within SDDL data that is usually not treated in pseudo-code and too many times left out of a design document. The constructs discussed in this section are based upon the module outline given previously in Table 3-1.

4.1 REQUIREMENTS DESCRIPTION

The requirements for a design are usually stated with respect to performance while the design is expressed in terms of data structure, task relationships and processing algorithms. In general, the requirements and the design will differ topologically and it will not be possible to embed the requirements in the program or data modules designed in response to these requirements.

Hence, a separate module construct for requirements is used. In a manner analogous to the modularization of data, program and task structure, the requirements for a system may be partitioned into hierarchical classes. Each class of requirements is contained in a REQUIREMENT module. The grouping of requirements into a particular REQUIREMENT module is done at the discretion of the designer. The governing criterion is the degree of REQUIREMENT module invocation that the designer wishes to employ. The module invocation keyword "REFINEMENT" is used to indicate a REQUIREMENT module containing a refinement of the requirements in the parent module.

In any REQUIREMENT module, all of the information is textual. The BLOCK and ENDBLOCK structure is allowed to aid the designer in automated indentation. Also, the SOURCE construct and the substructure for that construct is allowed to aid as a checklist to the designer as he/she describes the requirements.

A REQUIREMENT module can only point to another REQUIREMENT module. The construct used here is "REFINEMENT" and the meaning is that the referenced REQUIREMENT module contains a more detailed description of the requirements specified in the parent MODULE. REFINEMENT allows a structuring from a higher level requirement to a lower level requirement. A REQUIREMENT module can be pointed to by the construct SATISFIES. Figure 4-1 gives a sample REQUIREMENT and shows a use of the substructure under REQUIREMENTS.

4.2 REQUIREMENTS MODULARIZATION

In the case of process and data structures, there are a number of different module types. For example, PROGRAM, SUBROUTINE, FILE, and QUEUE. In the case of requirements, there is just one module type, namely, REQUIREMENT.

4.3 REQUIREMENTS ORGANIZATION

In any REQUIREMENT module, all of the information is textual. The BLOCK and ENDBLOCK structure is allowed to aid the designer in automated indentation. Also, the SOURCE construct and the substructure for that construct is allowed to aid as a checklist to the designer as he/she describes the requirements.

4.4 REQUIREMENTS INVOCATION

A REQUIREMENT module can only point to another REQUIREMENT module. The construct used here is "REFINEMENT" and the meaning is that the referenced REQUIREMENT module contains a more detailed description of the requirements specified in the parent MODULE. REFINEMENT allows a structuring from a higher level requirement to a lower level requirement.

```
REQUIREMENT REQ.GLL008

   SOURCE

      THE SOURCE (DOCUMENT) OF THIS REQUIREMENT IS ENTERED HERE

   VERSION

      THE VERSION NUMBER OR DATE IS ENTERED HERE

      UNFORTUNATELY THE INFORMATION CAN ONLY BE TEXTUAL
   STATUS

      A STATUS INDICATOR IS PLACED HERE.  THERE IS NO AUTOMATIC UPDATING OF
      THE INDICATION IN A GLOBAL FASHION

   SCOPE

      A SCOPE INDICATOR IS PLACED HERE.  THE PROJECT SHOULD DETERMINE A SET
      OF VALUES SUCH AS GLOBAL, LOCAL, HARD OR SOFT.

   PERFORMANCE.CONSTRAINT
      TEXTUAL INFORMATION ON THE PERFORMANCE CONSTRAINTS ARE ADDED HERE.

   REQUIREMENT.DESCRIPTION

      TEXTUAL INFORMATION IS PLACED HERE THAT DESCRIBES THE REQUIREMENT.

      REFINEMENT---------------------------------------------------------------------------------------->(****)
      THE REFINEMENT STATEMENT ALLOWS THE DESIGNER TO INDICATE THAT
      THIS REQUIREMENT IS A REFINEMENT OR FURTHER DELINEATION OF A
      HIGHER LEVEL REQUIREMENT.

   ENDSOURCE

ENDREQUIREMENT
```

Figure 4-1. Sample Requirement

4.5 DATA DESCRIPTION

The SDDL constructs that support data description are based upon the realization that data can be structured in a manner completely analogous to the structure provided for execution control structure. The basic assumption that has been made, is that all data items will be explicitly declared through the use of one or more SDDL data constructs. In addition, input/output of data can be thought of as viewing a collection of data elements through a window. The data elements of a particular data module currently on view through the window (the current record) can be utilized directly without being invoked by any special constructs such as READ or WRITE.

The reader is reminded that the SDDL constructs in this paper are oriented to the description of a detailed abstract design. Upon reading the material on data descriptions, the reader may first believe that physical implementation considerations are being described. That is not the case, even though the mapping from a construct such as "FILE" into a physical realization as an ISAM file is trivial. That such mappings may be straightforward is a deliberate consequence of the desire to provide the reader with a set of useful and complete SDDL constructs. The data structures described below are characterized by the abstract access methods that are supported for each particular data structure. The data organization techniques within any structure are identical (sequence, selection, replication).

The reader is reminded that the data description constructs represent an abstract view of the data and that in general, a physical collection of data may be presented in a variety of valid abstract forms. Often the quality of a design depends upon the insight of the designer in choosing the most useful approach to the organization of the programs and data. Further, the same data set may be described by two or more different abstract representations at different points in the processing of the data.

4.6 DATA MODULARIZATION

The data declarations are grouped into modules whose names imply the access characteristics of the enclosed data. The atomic elements within each module will be called "data declarations". A data declaration will consist of: the specification of a variable, a data module invocation (DETAILS) or a body of "data declarations" contained within a BLOCK construct. In general, an atomic element of data declaration, or datum, is a line of text starting with a variable name. Unless the name contains a special character that has been designated by the #MARK directive to be a non-delimiter, or it is enclosed in string delimiters (the default is quotes), the name is not cross referenced by SDDL. The datum may be followed by attribute information for that datum. For example:

```
Velocity.Vector          (A 3-dimensional vector with units KM/SEC)
```

At the level of abstract design given above, "Velocity.Vector" is an atomic element. Later, in the design process, this datum could be expanded into a structure.

Data modules are composed of collections of homogeneous components called "records". A data module is specified abstractly by: naming the module, specifying the module class by means of the key word, declaring the attributes of the underlying "records" and optionally providing supplemental parameters. The "record" provides a "window" through which the data module may be viewed. The access algorithms (the method by which the view is changed to another instance of a record) is implicit in the MODULE name and the MODULE INVOCATIONs used.

The following is a brief description of the types of data modules used. A "GROUP" is a hierarchical collection of data declarations, having the characteristics that there is only one copy of the record (it is not a multirecord structure) and access is accomplished without special program consideration. The implication with a set of data items that have been declared a GROUP is that these items are always "in view". Physical realizations are: local variables, FORTRAN Common and HAL/S Compool.

The following is an example of a GROUP named Velocity.Data.

```
GROUP Velocity.Vector          Units are KM/SEC
      Velocity.Vector.X
      Velocity.Vector.Y
      Velocity.Vector.Z
ENDGROUP
```

Note that "Velocity.Vector" is known to the SDDL processor and will be cross referenced. "Units are KM/SEC" is attribute information that will appear in the SDDL listing but is not processed. The three components of "Velocity.Vector" may be known to the SDDL processor (by virtue of the embedded period).

A "RECORD" is an instance of a GROUP. GROUP provides the data declaration and the use of the term "RECORD" allows one to discuss multiple examples. The constructs TABLE, FILE, STACK and QUEUE are collections of RECORDs structured in particular ways.

A "TABLE" is a randomly accessible collection of records (entries), indexed by an ordinal. (An n-tuple of ordinals is considered an ordinal.) The primitives with respect to table manipulation are "OPEN", "CLOSE", "READ", "WRITE", "GET", "PUT" and "CLEAR". Note that the number of records in a table is either fixed or defined at the point at which it is OPENed, and the CLEAR directive differs from that used with respect to FILEs in that it causes each record to be restored to a well-defined initialization state. Typically, this will involve zeroing numeric fields, blanking out character fields, etc.

The physical realization of the construct "TABLE" is often a FORTRAN direct access file. However, the construct described above may apply to a structure that may ultimately reside in memory, on a disk, as well as on a magnetic tape. An example is an array in Fortran.

The following is an example of a TABLE.

```
TABLE Velocity.Vector              Units are KM/SEC
      Velocity.Vector.X
      Velocity.Vector.Y
      Velocity.Vector.Z
TABLEPARAMETERS
     Velocity.Vector.Counter      Index for Table
     There are 100 entries in this table
ENDTABLE
```

In this particular example, the data structure used as an example for a GROUP has been extended to allow for many instances of the underlying group, where the indexing parameter is "Velocity.Vector.Counter".

A "FILE" is an ordered sequence of entities (usually called records), each analogous to a GROUP. The records of a file are indexed by a well ordered variable (usually called the access key). When the index is an integer, this corresponds to the concept of an array or an indexed direct access file and is more properly described as a TABLE. In the more general case where the index variable does not correspond to the concept of an ordinal, then a FILE corresponds to the concept of a keyed direct access data set. Only one element (record) of the file is presented to the program at a time. The primitive operators with respect to a FILE are: "OPEN", which makes the FILE available to the invoking program and unconditionally restores the file to the beginning; "CLOSE" which removes the file from accessability; "READ", which obtains the next sequential record if one is available; "WRITE", which merges a new record into the sequence of file records; "GET", which obtains the record, if one is available that corresponds to the value of the access key; "PUT", which merges a record corresponding to the key into the FILE; "DELETE", which purges the record corresponding to the key from the FILE and "CLEAR" which removes all of the records from the FILE. Note that WRITE differs from PUT, in that the generation of the appropriate key is assumed to be done by the underlying access method.

The physical realization of the construct "FILE" is usually a keyed index-sequential file, where the access method is vendor supplied. In those cases where no such vendor supplied access method exists, the implication is that a subsequent refinement of the design will contain the algorithm and the underlying physical data structure (e.g., a linked list contained within a table). The example given for TABLE immediately above is also suitable for FILE. A FILE differs from a TABLE in that the number of records is not fixed, the access algorithms are more complex and versatile, and records may be added or deleted.

The following is an example of an on-line payroll data base, where the records may be retrieved by employee name, social security number or employee number.

ORIGINAL PAGE IS
OF POOR QUALITY

```
FILE PAYROLL
     PAYROLL.EMPLOYEE.NO 1  (*Employee number, 0 to 10,000)
     PAYROLL.NAME C40  (*Name of employee,  up to 40 characters)
     PAYROLL.SOCIAL.SECURITY.NO 1  (*9-digit social security number)
     PAYROLL.JOB.CLASS C  (*Job classification, 1 character)
     DETAILS PAYROLL.GOVT.REQUIRED.DATA
     DETAILS PAYROLL.YEAR.TO.DATE.TOTALS
FILEPARAMS
     This file is indexed in PAYROLL.EMPLOYEE.NO, PAYROLL.NAME and
     PAYROLL.SOCIAL.SECURITY.NO, therefore records may be accessed by
     any of the three "keys".
     The file contains a maximum of 5,000 entries
ENDFILE
```

A "STACK" is a collection of records that are created in order and destructively read back in reverse order. The primitives for stack manipulation are "PUSH", "POP" and "CLEAR". Note that no index is used in the description of a "STACK". In the case of a "STACK", it is first in, last out. The construct OPEN "creates" the STACK while the construct CLOSE removes the STACK from access by the program.

The following is an example of a STACK named "Daily.Menu".

```
STACK Daily.Menu
     Menu.Indicator (1=Breakfast, 2=Brunch, 3=Lunch, 4=Dinner,
                     5=Supper, 6=Snack)
     Meal.Plan
ENDSTACK
```

A "QUEUE" is a collection of records, ordered by priority (usually the chronology of creation) and retrieved (typically in a destructive manner) in order of priority. The actual priority mechanism is established during the physical realization of the particular QUEUE. This implies that the priority mechanism for one QUEUE may be different than the priority mechanism for a second QUEUE. When defining such abstract QUEUEs, the designer must be careful to define only those abstractions that are supportable in the target implementation. The primitives with respect to queue manipulation are "CLEAR", "OPEN", "CLOSE", "ENQUEUE" AND "DEQUEUE". A chronological "QUEUE" has the property of first in, first out, and is assumed in those cases where no priority mechanism is explicitly specified. The physical realization of a "QUEUE" is many times an input card file or a printed listing.

A "TEMPLATE" is a collection of data passed across the interface between two program modules, where the data items must have been declared elsewhere. In this particular case, an interface is a major boundary such as the line between two independently created programs or between two different programs operating on two physically different computers. The construct TEMPLATE allows the designer to highlight what is being passed across the interface. The data items of the TEMPLATE may be complete data structures such as a GROUP or a FILE or they may be portions of data structures.

The following is an example of a TEMPLATE named Interface.1.

```
TEMPLATE Interface.1
     Time      (the system time of the transfer of data, units - GMT)
     Details Velocity.Data
ENDTEMPLATE
```

4.7 DATA ORGANIZATION

Within each module of data declaration, the hierarchy of organization and the details are presented through a combination of data items, module reference and block constructs.

A data item (atomic element) is a unit of data representation that requires (relative to the level of design under consideration) no further decomposition. Examples are: an integer variable, the two real numbers corresponding to a single complex number in an environment supporting complex arithmetic, a coordinate transformation matrix or a character string. There is no keyword associated with data items in any data structure.

Hierarchical data constructs of sequence, selection and iteration are used. At a particular level in a data structure, all data items must be of the same type. Further, the ordering of the data is top-to-bottom, left to right. In SDDL the default option is SEQUENCE. The implication here is that all data items mentioned appear and they appear in the order in which they are listed. The "REPLICATION" construct defines a set of data declarations that may be replicated zero or more times. Note that the constructs of TABLE, STACK, and QUEUE have an implied REPLICATION as a part of the construct. Figure 4-2 gives an example of the use of REPLICATION in a GROUP structure.

```
LINE

  51 GROUP DATA.RECORDS                                          000
  52    NO.OF.DATA.RECORDS                                       000
  53    REPLICATION ON NO.OF.DATA.RECORDS                        000
  54       DATA.RECORD.ID                                        000
  55       SPACE.CRAFT.POSITION                                  000
  56       SPACE.CRAFT.VELOCITY                                  000
  57       NO.OF.SCIENCE.EXPERIMENTS                             000
  58       REPLICATION ON NO.OF.SCIENCE.EXPERIMENTS              000
  59          SCIENCE.EXPERIMENT.DATA                            000
  61 ENDGROUP                                                    000
```

Figure 4-2. Replication

The "SELECTION" construct defines a set of candidate data declarations, one and only one of which must be present. The substructure within SELECTION is ALTERNATE. The convention is that this substructure must be used. Figures 4-3a and 4-3b show uses of the constructs for structuring data.

ORIGINAL PAGE IS
OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

```
GROUP SDR
   DETAILS TDRID----------------------------------------------------->( 9)
   DETAILS SDR-HDR--------------------------------------------------->( 7)
   SELECTION ON DATA.TYPE = ' '
   <--ALTERNATE SDR-SUBHDR
      DETAILS SDR-SUBHDR--------------------------------------------->( 7)
   <--ALTERNATE NULL
   ENDSELECTION
   SELECTION ON DATA.TYPE = ' '
   <--ALTERNATE DECOM-MAP
      DETAILS DECOM-MAP---------------------------------------------->( )
   <--ALTERNATE ENGINEERING
      DETAILS ENGINEERING-------------------------------------------->( )
   <--ALTERNATE GENERAL-SCIENCE
      DETAILS GENERAL-SCIENCE---------------------------------------->( )
   <--ALTERNATE IMAGING
      DETAILS IMAGING------------------------------------------------>( )
   <--ALTERNATE PLAY-BACK
      DETAILS PLAY-BACK---------------------------------------------->( )
   <--ALTERNATE NULL
   ENDSELECTION
   DETAILS SDR-DATA------------------------------------------------->( 7)
ENDGROUP
```

Figure 4-3a. Sample Data Structure

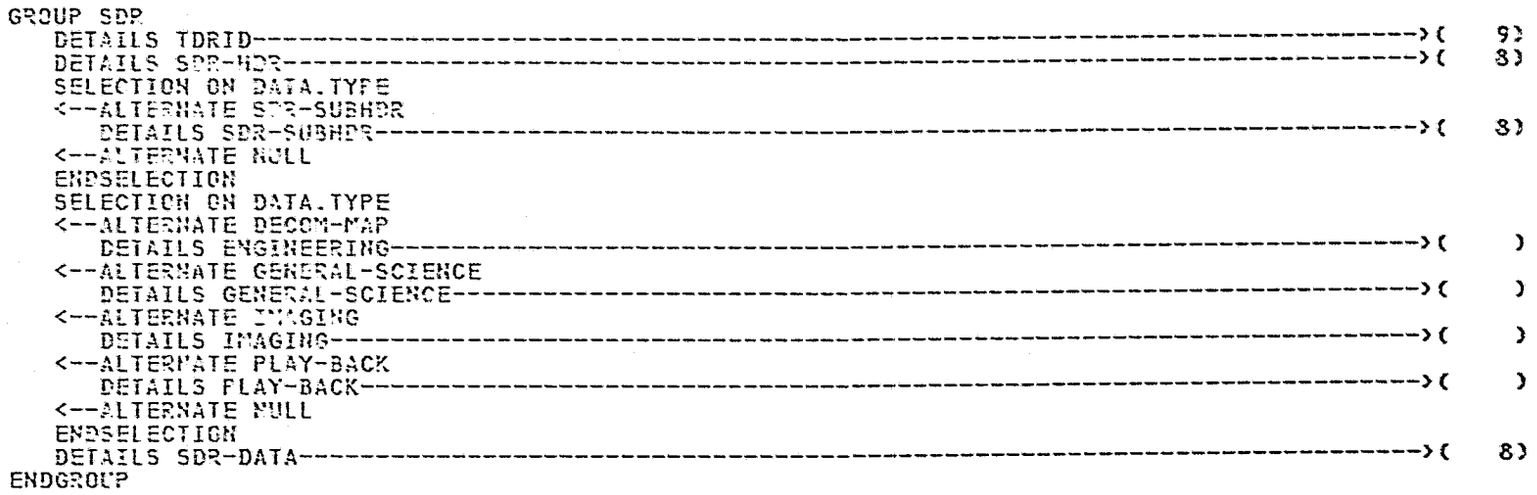

Figure 4-3b. Sample Data Structure (1 of 2)

```
GROUP TDRID

   ************************************************************************
   *                                                                      *
   *  THE HEADER RECORD CONSISTS OF TWO WORDS AND CONTAINS THE            *
   *          RECORD IDENTIFICATION.                                      *
   *                                                                      *
   *             WORDS 0-1      BITS 0-15                                 *
   *                                                                      *
   ************************************************************************
   BLOCK

            TDPRJ

            ***************************************************************
            *  PROJECT - CONTAINS THE PROJECT IDENTIFICATION              *
            *     OF THE DATA CONTAINED IN THIS RECORD                    *
            *         0 = MJS                                             *
            *                                                             *
            *              WORD  0        BITS 0-3                        *
            *                                                             *
            ***************************************************************
            TDVAL

            ***************************************************************
            *  VALID DATA - INDICATES THE PRESENCE OF DATA IN             *
            *     THE RECORD.  THIS FIELD IS USED TO                      *
            *      DIFFERENTIATE BETWEEN REAL DATA AND DATA BREAK         *
            *     CONDITIONS IN THE TELEMETRY DATA                        *
            *        0 = VALID DATA EXISTS                                *
            *        1 = NO DATA  (DATA BREAK IF TDRLG=1)                 *
            *                     (FLUSH      IF TDRLG=5)                 *
            *                                                             *
            *              WORD  0        BIT 4                           *
            *                                                             *
            ***************************************************************
            TDSCID

            ***************************************************************
            *  SPACECRAFT ID                                              *
            *          0 = 32    1 = 31    2 = PTM    3 = UNKNOWN         *
            *          4 = 41    5 = 42                                   *
            *                                                             *
            *              WORD  0        BITS 5-7                        *
            *                                                             *
            ***************************************************************
         ENDBLOCK

ENDGROUP
```

Figure 4-3b. Sample Data Structure (2 of 2)

ORIGINAL
OF POOR QUALITY

The module invocation construct "DETAILS" is provided solely for the sake of convenience. It permits a body of data declarations to be named, enclosed in a module construct, placed elsewhere in the document and referenced by the keyword "DETAILS" followed by the name of the module. In addition, it allows one to show, in a convenient manner, part of a module that has different access characteristics, e.g., a table embedded in a file record. Note that this construct admits to a convenient design representation of a concept that may be extremely difficult to implement if access characteristics are mixed.

In some of the examples given above, the construct DETAILS has been used. Note that SDDL produces a page number on the right hand side that indicates where the "detailed" module may be found.

4.8 DATA INVOCATION

The only data module invocation permitted within a data module declaration is "DETAILS". It may be used with any of the six data constructs. Note that invocation such as OPEN and CLOSE are used by procedural modules and provide a mapping from such a procedural module to the data structure that is to be opened or closed. The constructs OPEN or CLOSED may be used with any DATA module other than GROUP and TEMPLATE. The physical realization of the construct depends upon the type of the DATA module and the particular implementation.

4.9 PROGRAM DESCRIPTION

The procedural aspects (control processes) of a software system are described in terms of program modules and a set of constructs to define the flow of execution within a program module. For the purpose of this document, the atomic elements of the "executable" portion of a software design will be called "execution directives". An "execution directive" will consist of an algorithmic operator (free form text), a module invocation or a body of execution directives contained within an SDDL block construct.

4.10 PROGRAM MODULARIZATION

The executable portion of a software design is partitioned into modules, typed by the method of invocation and the method by which the interface between control process modules is effected. A "PROGRAM" is a hierarchical collection of execution directives. Programs are invoked only through the operating system. A "PROCEDURE" is a hierarchical collection of execution directives that are local to a parent execution module (a program subroutine or function). A "SUBROUTINE" is a hierarchical collection of execution directives that may be invoked from any other execution module. A "FUNCTION" is a special kind of subroutine that returns a scalar to the calling module. A function also differs from a subroutine in that it is typically invoked without the use of an explicit module invocation.

4.11 PROGRAM FLOW CONSTRUCTS

The flow of execution within an execution module is specified by means of hierarchical constructs and execution directives. An "Operation" is a particular type of an execution directive. It is a free form text statement. "Operations" may be thought of as algorithms that require no further decomposition relative to the level of design under consideration. There is no keyword associated with these directives. The "IF" construct describes one or more bodies of execution directives, where the execution of at most one is contingent upon the evaluation at execution time of a predicate test. The "SELECT" construct provides for the execution of at most one of a set of code bodies, dependent upon the value at execution time of a scalar variable. The "LOOP" construct provides for the execution iteration of a code body. The "PERFORM" construct provides for the conditional execution of a set of bodies of execution directives where the the blockwise order of execution is not a factor in the abstract design.

4.12 PROGRAM INVOCATION

Program invocation constructs allow a pointer to be established between a program module and a program module, requirements module or a task module. The construct "CALL" is to be used by a PROGRAM, SUBROUTINE or FUNCTION module to point to another SUBROUTINE or FUNCTION. The construct "DO" is to be used to point to a PROCEDURE. The assumed entry point of the program module that is being referenced by a "CALL" or "DO" is always the first executable statement of that program module. The construct "EXECUTE" is to be used by a program to point to another program. In this way, control can be transfered between two different programs.

The construct "SATISFIES" is used by any module to invoke a requirements module. Its use is only for the purpose of providing requirments traceability.

4.13 DATA USAGE

Data invocation constructs allow a pointer to be established between a process module and a data module. The constructs that are used are: OPEN, CLOSE, READ, WRITE, GET, PUT, DELETE, CLEAR, ENQUEUE, DEQUEUE, PUSH, POP, ACCESSES, CREATES, and MODIFIES.

The construct MODIFIES provides the facility for a designer to indicate that the contents of the object data structure are modified. The object data structure may be used as well. It may apply to any data structure.

4.14 TASK DESCRIPTION

Programming is the means by which a general purpose machine (computer) is converted into a special purpose machine. In some designs, it may be useful or even necessary to partition the functions to be performed into a set of independent abstract machines. The most straightforward instance is the case in which two or more computers are involved and the

separation of the functions is physical as well as abstract. In other cases, this separation is effected through the facilities of a modern operating system. In either case, such an "abstract" machine will be called a TASK. A TASK has the characteristics that it "uses" executable software elements (programs) which define its function, and it "owns" data, which defines the scope of its machine state. Consequently, the description of the operation of a task will contain no executable directives but it will be described functionally by reference to the program that is owned by the task. This concept of TASK is most suitable for designs in which the functioning of independent elements is loosely coupled. Under execution flow we have already introduced constructs that may be used for the description of closely coupled independent activities. In some environments, a tasks fate may be contingent upon the fate of some other task. In this case the task is said to be dependent. The specification of a task consists of the description of the characteristics of the abstract "machine" comprising the declaration of the programs, data and dependent task relationships. It should be noted that not all operating systems support an environment in which the abstract design using task relationships can be physically realized. If the operating system supports dynamic tasking, then there will be constructs in the operating system that will be the physical realization of the SDDL constructs ATTACH and EXECUTE.

4.15 TASK MODULARIZATION

Because of the importance of a task, all of its components (subtasks) must be known to the SDDL processor. The specification of the TASK will include all those characteristics of the abstract machine that are independent of the program that is executed.

The following is an example of the abstract machine that supports communication with devices attached to a CPU by means of the DMA channels:

```
TASK DMA Interrupt Processor
     This TASK services all DMA interrupts.
     This TASK operates with all interrupts except machine check
     inhibited.
     ATTACHMENT REENTRANT-DMA-INTERRUPT-HANDLER
     OWNERSHIP DMA.TASK.CONTROL.BLOCK
     OWNERSHIP DMA.CHANNEL.DEFINITION.TABLE
     OWNERSHIP DEVICE.CONTROL.BLOCK
ENDTASK
```

Task Control Constructs

A task contains no executable directives. Hence, no control constructs are required.

ORIGINAL PAGE IS
OF POOR QUALITY

Task Invocation

A task can point to other subtasks by the construct "SUBORDINATION". A task can point to any data module by the construct "OWNERSHIP". A task can point to any program module by the construct "ATTACHMENT".

4.16 TASK CONTROL CONSTRUCTS

A task contains no executable directives. Hence, no control constructs are required.

4.17 TASK INVOCATION

A task can point to other subtasks by the construct "SUBORDINATION". A task can point to any data module by the construct "OWNERSHIP". A task can point to any program module by the construct "ATTACHMENT".

4.18 DOCUMENTATION DESCRIPTION

SDDL constructs have been introduced to aid in the creation of documentation. These constructs match the module outline given in Table 3-1. These constructs are at the BLOCK level and hence, may be used within a data module, a process module, or a task module. They can be used within a requirements module but they are not suitable for such use.

The invoking construct is ABSTRACT. The constructs which give substructure within the abstract block match the outline descriptors from Table 3-1. The substructure constructs are:

```
ALLOCATED.REQUIREMENTS
DESIGN.DESCRIPTION
AFFECTED.MODULES
OPERATING.ENVIRONMENT
SYSTEM.PARAMETERS
DATA.SPECIFICATION
PROCESS.SPECIFICATION
DATA.ORGANIZATION
CONTROL.FLOW
```

The construct SATISFIES under ALLOCATED.REQUIREMENTS allows the designer when describing a PROCESS module, DATA module, or TASK module to point to the REQUIREMENTS module that this module satisfies.

The construct AFFECTED.MODULES provides a place for the designer to record all of the invocations that this module participates in. Such information is automatically obtained from the SDDL cross reference listing and the module tree listing.

A set of SDDL constructs have been introduced to aid in the description of a REQUIREMENT module. The invoking construct is SOURCE. The substructure constructs are:

VERSION
STATUS
SCOPE
PERFORMANCE.CONSTRAINT
REQUIREMENT.DESCRIPTION

These constructs provide prompts to the designer as the requirements are documented. An example of their use is given in Figure 4-4.

```
REQUIREMENT R001
   THE PROCEDURE SHALL BE ABLE TO READ ONE LINE AT A TIME
   REFINEMENT R00A---------------------------------------------------------------------------->(
ENDREQUIREMENT



PROCEDURE GET-NEXT-SOURCE-STATEMENT

   ABSTRACT

      THIS PROCEDURE RETURNS ONE SOURCE RECORD IN SOURCE.RECORD.
   <--DESIGN.DESCRIPTION
      THE FILE
      FROM WHICH THE RECORD IS OBTAINED, AS WELL AS ITS LOCATION WITHIN
      THAT FILE, IS UNDER THE CONTROL OF THE RECURSION.CONTROL.STACK.

   ALLOCATED.REQUIREMENTS
      SATISFIES R001----------------------------------------------------------------------------->(  10)
   CONTROL.FLOW

      CLEAR CURRENT.SOURCE.STATEMENT
      IF (RECURSION.INDEX = 0) (*TEST FOR PRIMARY SOURCE FILE)
         DO FETCH-NEXT-SOURCE-RECORD------------------------------------------------------------->(     )
<--------EXIT PROCEDURE IF (SOURCEIN.EOF)
         MOVE SOURCEIN.RECORD TO SOURCE.RECORD
      ELSE
         THE FILE IS DEFINED BY RECURSION.FILENAME
         READ USER_MACRO_DEFINITION_FILE--------------------------------------------------------->(     )
         IF (MACDEF.EOF)
            ERROR.NUMBER := ERROR.0803
            DO OUTPUT-ERROR-MESSAGE------------------------------------------------------------->(     )
         ENDIF
         INCREMENT RECURSION.RECNO
      ENDIF
   ENDABSTRACT
ENDPROCEDURE
```

Figure 4-4. Use of Abstract

ORIGINAL PAGE IS OF POOR QUALITY

SECTION 5

DEFINITION OF CDS

The Galileo spacecraft is a Jupiter orbiter scheduled for launch in 1984. The Command and Data Subsystem (CDS) provides the control function for the Galileo spacecraft. The software design for the first delivery of CDS software was done using SDDL. The design is for the software that makes the CDS hardware perform as an interpretive machine. Programs for that machine are stored sequences of commands.

The CDS is a bus-oriented distributed data network which forms the core of the spacecraft distributed system. The extension of the CDS bus to external subsystems provides the primary communication path between the CDS and other subsystems.

The SDDL description of the detailed design for CDS is given in over eighty pages of SDDL listings. To describe the detailed software design, there are 39 modules of control flow and 8 modules of data description. Figure 5-1 is the detailed design for the top level executive. Note that not all of the eight dimensions (see Table 3-1) were used to describe the example.

Figure 5-2 is an example of a function description in SDDL. It is the TIME-TO-EXECUTE function.

Because of the specialized nature of the CDS software, the only data construct that was used was that for GROUP. This was because of the special I/O characteristics of the CDS hardware. Figure 5-3 is a description of the GROUP entitled HLM-SYSTEM-START-TE. Note that the DATA.SPECIFICATION entry was used to indicate the initial values of this data structure when the CDS software is cold started. Another example of a GROUP data structure is given in Figure 5-4. Note that under DATA.SPECIFICATION, declare statements for the individual bits and bytes are given. These declare statements are passed as text by the SDDL processor.

Another example of a GROUP data structure is given in Figure 5-5. In this case, a substructure is added under the DATA.SPECIFICATION entry.

One of the major outputs of the SDDL processor is the module reference tree. It gives the full network structure for all of the module invocations. Figure 5-6 is the first part of the full module invocation tree for the CDS software.

```
PROGRAM EXEC
   ABSTRACT
   DESIGN.DESCRIPTION
      ************************************************************************************************************************
      * THE EXECUTIVE IS THE MAIN CONTROL PROCESS FOR EACH PROCESSOR                                                         *
      * MODULE IN THE CDS. IT PERFORMS SOFTWARE INITIALIZATION UPON ENTRY                                                    *
      * AND LOOPS FOREVER PERFORMING ADDITIONAL FUNCTIONS. THESE FUNCTIONS                                                   *
      * ARE: INSURE THAT PROCESSING HAS NOT EXCEEDED AVAILABLE TIME; WAIT                                                    *
      * FOR AN RTI WHILE COLLECTING WORST CASE IDLE TIME PER RTI; INCREMENT                                                  *
      * THE SPACECRAFT CLOCK; CALL THE BUS MANAGER; AND PROCESS ACTIVE                                                       *
      * TIME/EVENT REGIONS. THE EXECUTIVE IS INTERRUPTED BY RTI-PROCESSOR                                                    *
      * AND USES A BOOLEAN FUNCTION TIME-TO-EXECUTE IN PROCESSING ACTIVE                                                     *
      * TIME/EVENT REGIONS.                                                                                                  *
      ************************************************************************************************************************
   DATA.SPECIFICATION
      MODIFIES CONTROL-TABLE---------------------------------------------------------------------------------------->(  58)
      MODIFIES CLOCK------------------------------------------------------------------------------------------------>(  60)
      MODIFIES REGISTER--------------------------------------------------------------------------------------------->(  57)
      MODIFIES CCDC-QUEUE------------------------------------------------------------------------------------------->(  65)
   OPERATING.ENVIRONMENT
      APRAM IN BOTH LLM'S AND HLM'S
   CONTROL.FLOW
      RESET REGISTER(INTERRUPT)
      INITIALIZE CONTROL-TABLE TO ALL CONTROL-TABLE SLOTS NOT-ACTIVE
      START SYS-START-TE IN RTI 0 SEGMENT OF CONTROL-TABLE
      INITIALIZE CLOCK TO AN ABSOLUTE TIME OF 0
      INITIALIZE FRONT AND REAR TO 1 /*COMMENTED OUT IN HLM LOAD*/
      DO FOREVER
         IF (REGISTER(INTERRUPT) IS SET) THEN
            CALL ERROR(SATURATION)------------------------------------------------------------------------------->(  13)
         ENDIF
         CLEAR IDLE-COUNTER
         DO WHILE (REGISTER(INTERRUPT) IS NOT SET) /* EXEC IS INTERRUPTED FROM THIS LOOP */
            INCREMENT IDLE-COUNTER
         ENDDO
         RESET REGISTER(INTERRUPT)
         IF (REGISTER(RTIO-IDLE + CLOCK.RTI) > IDLE-COUNTER) THEN
            REGISTER(RTIO-IDLE +CLOCK.RTI) = IDLE-COUNTER
         ENDIF
         INCREMENT CLOCK
         CALL BUS-MANAGER /*COMMENTED OUT IN LLM LOAD*/------------------------------------------------------->(  12)
         DO FOR SLOTPTR = BEGINING OF CLOCK.RTI SEGMENT TO END OF CLOCK.RTI SEGME                                      NT
            IF (CONTROL-TABLE.TIME-CODE(SLOTPTR) NOT EQUAL NOT-ACTIVE) THEN
               DO WHILE TIME-TO-EXECUTE
                  CALL HDFCODE /*COMMENTED OUT IN LLM LOAD*/--------------------------------------------------->(   8)
                  CALL LDFCODE /*COMMENTED OUT IN HLM LOAD*/--------------------------------------------------->(  10)
                  IF (CONTROL-TABLE.TIME-CODE(SLOTPTR) = RELATIVE MINOR FRAME OR RELATIVE MAJOR FRAME) THEN
                     CONTROL-TABLE.DELTA-TIME(SLOTPTR) = 0
                  ENDIF
               ENDDO
            ENDIF
         ENDDO
      ENDDO
   ENDABSTRACT
ENDPROGRAM
```

Figure 5-1. Detailed Design - Top Level EXECUTIVE

ORIGINAL PAGE IS OF POOR QUALITY

```
FUNCTION TIME-TO-EXECUTE
   ABSTRACT
   DESIGN.DESCRIPTION
      ************************************************************************************************************************
      * THIS BOOLEAN FUNCTION IS USED WITH EXEC TO DETERMINE IF IT IS                                                        *
      * TIME TO EXECUTE A PSEUDO EVENT IN AN ACTIVE TIME/EVENT REGION.                                                       *
      * IN ADDITION, THIS FUNCTION INCREMENTS THE CONTROL TABLE                                                              *
      * DELTA TIME DEPENDING ON TIME CODE. THE CONTROL TABLE POINTER                                                         *
      * IS ALWAYS UNDISTURBED UPON RETURN FALSE AND IS INCREMENTED                                                           *
      * TO POINT TO THE PSEUDO EVENT ID UPON RETURN TRUE.                                                                    *
      ************************************************************************************************************************
   DATA.SPECIFICATION
      MODIFIES CONTROL-TABLE----------------------------------------------------------------------------------------->(  58)
      ACCESSES CLOCK------------------------------------------------------------------------------------------------>(  60)
   OPERATING.ENVIRONMENT
      MUST BE USED IN WPRAM WITH EXEC
   CONTROL.FLOW
      DOCASE CONTROL-TABLE.TIME-CODE(SLOTPTR)
      CASE  NOT-ACTIVE
         RETURN FALSE
      CASE  RELATIVE MINOR FRAME
         IF (CONTROL-TABLE.DELTA-TIME(SLOTPTR) >= VALUE POINTED TO BY CONTROL-TABLE.POINTER(SLOTPTR)
            INCREMENT CONTROL-TABLE.POINTER(SLOTPTR) BY ONE
            RETURN TRUE
         ELSE
            INCREMENT CONTROL-TABLE.DELTA-TIME(SLOTPTR) BY ONE
            RETURN FALSE
         ENDIF
      CASE  ABSOLUTE MINOR FRAME
         IF (VALUE POINTED TO BY CONTROL-TABLE.POINTER(SLOTPTR) = CLOCK.MINOR-FRAME) THEN
            INCREMENT CONTROL-TABLE.POINTER(SLOTPTR) BY ONE
            RETURN TRUE
         ELSE
            RETURN FALSE
         ENDIF
      CASE  RELATIVE MAJOR FRAME
         IF (CLOCK.MINOR-FRAME = 0) THEN
            IF (CONTROL-TABLE.DELTA-TIME(SLOTPTR) >= VALUE POINTED TO BY CONTROL-TABLE.POINTER(SLOTPTR)) THEN
               INCREMENT CONTROL-TABLE.POINTER(SLOTPTR) BY ONE
               RETURN TRUE
            ELSE
               INCREMENT CONTROL-TABLE.DELTA-TIME(SLOTPTR) BY ONE
               RETURN FALSE
            ENDIF
         ELSE
            RETURN FALSE
         ENDIF
      CASE  ABSOLUTE TIME
         IF (CLOCK.MAJOR-FRAME(1) = VALUE POINTED TO BY CONTROL-TABLE.POINTER(SLOTPTR)) THEN
            IF (CLOCK.MAJOR-FRAME(2) = VALUE POINTED TO BY CONTROL-TABLE.POINTER(SLOTPTR)+1) THEN
               IF (CLOCK.MAJOR-FRAME(3) = VALUE POINTED TO BY CONTROL-TABLE.POINTER(SLOTPTR)+2) THEN
                  IF (CLOCK.MINOR-FRAME = VALUE POINTED TO BY CONTROL-TABLE.POINTER(SLOTPTR)+3) THEN
                     INCREMENT CONTROL-TABLE.POINTER(SLOTPTR) BY 4
                     RETURN TRUE
                  ELSE
                     RETURN FALSE
```

Figure 5-2a. Time to Execute Function

```
                ENDIF
             ELSE
                RETURN FALSE
             ENDIF
          ELSE
             RETURN FALSE
          ENDIF
       ELSE
          RETURN FALSE
       ENDIF
    CASE  IMMEDIATE
       RETURN TRUE
    CASE ALL OTHERS
       CALL ERROR(BAD-TIME-CODE)----------------------------------------------------------------------------------->(  13)
    ENDCASE
 ENDABSTRACT
ENDFUNCTION
```


Figure 5-2b. (Continued)

```
GROUP HLM-SYSTEM-START-TE
   ABSTRACT
   DESIGN.DESCRIPTION
      ************************************************************************************************************************
      * THIS TE-REGION INITIALIZES THE CONTROL-TABLE OF THE HLMS                                                              *
      * TO A KNOWN STATE AFTER RESET.                                                                                        *
      ************************************************************************************************************************
   OPERATING.ENVIRONMENT
      HLM #PRAM
      STARTS IN RTI 0
   DATA.SPECIFICATION
      DELTA-TIME = 0
      ID = TE-START
      ADDRESS = ADDRESS OF HCMD-TE
      TIME CODE = RELATIVE MINOR FRAME
      RTI = 3, USE = 15
      DELTA-TIME = 0
      ID = EOR
   ENDABSTRACT
ENDGROUP
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 5-3. HLM-SYSTEM-START-TE GROUP

ORIGINAL PAGE IS
OF POOR QUALITY

```
GROUP MIOSELECT
   ABSTRACT
   OPERATING.ENVIRONMENT
      MUST BE LOADED AT FF00 IN MLM
   DESIGN.DESCRIPTION
      ************************************************************************************************************************
      *   THE FOLLOWING GROUP MAPS ONTO THE HIGH LEVEL                                                                       *
      *   MODULE I/O SELECTS.                                                                                                *
      *                                                                                                                      *
      *    IOSL0 BIT(8)       /*ALIAS OF 'RESET EF3'*/                                                                     *
      *                  BIT(1)  BA TRANSACTION PARITY ERROR                                                                 *
      *                  BIT(2)  BA BUS PARITY ERROR                                                                         *
      *                  BIT(3)  MCD PARITY ERROR                                                                            *
      *                  BIT(4)  MICROPROCESSOR SYNC ERROR                                                                   *
      *                  BIT(5)  MLM POR FLAG      /*ALIAS OF 'EF3'*/                                                        *
      *                  BIT(6)  MLM KEEP ALIVE FLAG                                                                         *
      *                  BIT(7)  SELF TEST FAIL STATUS                                                                       *
      *                  BIT(8)  BUS OVERRUN STATUS                                                                          *
      *    IOSL1 BIT(8)                                                                                                      *
      *                  BIT(1)  MCD WRITE PROTECT ERROR                                                                     *
      *                  BIT(2)  MP WRITE PROTECT ERROR                                                                      *
      *                  BIT(3)  BA WRITE PROTECT ERROR                                                                      *
      *                  BIT(4)  MPLD STATUS                                                                                 *
      *                  BIT(5)  BC MEMORY READ PARITY ERROR                                                                 *
      *                  BIT(6)  BA MEMORY READ PARITY ERROR                                                                 *
      *                  BIT(7)  MP MEMORY READ PARITY ERROR                                                                 *
      *     IOSL2 BIT(8)  BIT(1)  POR STATUS 0                                                                               *
      *                  BIT(2)       POR STATUS 1                                                                          *
      *                  BIT(3)  POR STATUS 2                                                                                *
      *                  BIT(4)  POR STATUS 3                                                                                *
      *                  BIT(5)  TIMING CHAIN SEL STATUS 4                                                                   *
      *                  BIT(6)  TIMING CHAIN SEL STATUS 5                                                                   *
      *                  BIT(7)  TIMING CHAIN SEL STATUS 6                                                                   *
      *                  BIT(8)  TIMING CHAIN SEL STATUS 7                                                                   *
      *    IOSL3 BYTE       SPARE                                                                                            *
      *    IOSL4 BYTE       RESET EF1                                                                                        *
      *    IOSL5 BYTE       RESET EF2                                                                                        *
      *    IOSL6 BYTE       SELF-ID                                                                                          *
      *    IOSL7 BYTE       SELF TEST 1                                                                                      *
      *    IOSL8 BYTE       SELF TEST 2                                                                                      *
      *    IOSL9 BYTE       SELF TEST 3                                                                                      *
      *    IOSL10 INTEGER   BUS CONTROLLER                                                                                   *
      ************************************************************************************************************************
   DATA.SPECIFICATION
      DECLARE IOSL0 BIT(8)
      DECLARE IOSL1 BIT(8)
      DECLARE IOSL2 BIT(8)
      DECLARE IOSL3 BYTE
      DECLARE IOSL4 BYTE
      DECLARE IOSL5 BYTE
      DECLARE IOSL6 BYTE
      DECLARE IOSL7 BYTE
      DECLARE IOSL8 BYTE
      DECLARE IOSL9 BYTE
      DECLARE IOSL10 INTEGER
   ENDABSTRACT
```

Figure 5-4. DATA SPECIFICATION GROUP

```
GROUP CLOCK
   ABSTRACT
   DESIGN.DESCRIPTION
      ************************************************************************************************************
      *            DATA FIELD                BYTES                                                               *
      *      --------------------------                                                                          *
      *      |  CLOCK.MAJOR-FRAME(1)  |      1                                                                   *
      *      --------------------------                                                                          *
      *      |  CLOCK.MAJOR-FRAME(2)  |      1                                                                   *
      *      --------------------------                                                                          *
      *      |  CLOCK.MAJOR-FRAME(3)  |      1                                                                   *
      *      --------------------------                                                                          *
      *      |  CLOCK.MINOR-FRAME     |      1                                                                   *
      *      --------------------------                                                                          *
      *      |  CLOCK.RTI             |      1                                                                   *
      *      --------------------------                                                                          *
      ************************************************************************************************************
   DATA.SPECIFICATION
      DECLARE CLOCK TIME-STRUCTURE
      STRUCTURE TIME
         1 CONSTANT INITIAL C6 HEX
         1 MAJOR-FRAME ARRAY(3) BYTE
         1 MINOR-FRAME BYTE
         1 RTI BYTE
      ENDSTRUCTURE
   OPERATING.ENVIRONMENT
      LLM AND HLM RAM
   ENDABSTRACT
ENDGROUP
```

Figure 5-5. DATA SPECIFICATION GROUP (Added Substructure)

```
    ***************  MODULE REFERENCE TREE  ******
LN  PAGE
 1     1  RTI-PROCESSOR
 2    53  . MIOSELECT
 3    55  . LIOSELECT
 4    57  . REGISTER
 5     3  . RESET
 6    53  . . MIOSELECT
 7    55  . . LIOSELECT
 8     3  . . RESET
 9           ** EXPANDED ON LINE   5
10     5  . . EXEC
11    5A  . . . CONTROL-TABLE
12    60  . . . CLOCK
13    57  . . . REGISTER
14    65  . . . CCDC-QUEUE
15    13  . . . ERROR
16    57  . . . . REGISTER
17    5A  . . . . CONTROL-TABLE
18     3  . . . . RESET
19           ** EXPANDED ON LINE   5
20    12  . . . BUS-MANAGER
21    60  . . . . CLOCK
22    61  . . . . BUS-CONTROL
23    53  . . . . MIOSELECT
24     8  . . . MDECODE
25    5A  . . . . CONTROL-TABLE
26    22  . . . . NOP
27    5A  . . . . . CONTROL-TABLE
28    42  . . . . MLM-CCDC
29    60  . . . . . CLOCK
30    64  . . . . . BUSTABLE
31    5A  . . . . . CONTROL-TABLE
32    13  . . . . . ERROR
33           ** EXPANDED ON LINE  15
34    40  . . . . MUS-TRANSACTION
35    60  . . . . . CLOCK
36    64  . . . . . BUSTABLE
37    61  . . . . . BUS-CONTROL
38    5A  . . . . . CONTROL-TABLE
39    36  . . . . MLM-GET-CMD
40    5A  . . . . . CONTROL-TABLE
41    63  . . . . . MLM-COMMAND-BUFFER
42    57  . . . . . REGISTER
43    53  . . . . . MIOSELECT
44    13  . . . . . ERROR
45           ** EXPANDED ON LINE  15
46    3A  . . . . . UIX
47    33  . . . . TE-START
48    5A  . . . . . CONTROL-TABLE
49    13  . . . . . ERROR
50           ** EXPANDED ON LINE  15
51    35  . . . . EOR
52    5A  . . . . . CONTROL-TABLE
53    23  . . . . JUMP
54    5A  . . . . . CONTROL-TABLE
55    24  . . . . JVEQ
```

Figure 5-6. CDS Software (Module Invoca ee)

ORIGINAL PAGE IS
OF POOR QUALITY

SECTION 6

EXPERIENCE USING SDDL ON CDS DETAILED DESIGN

In general, the experience gained with SDDL while designing the CDS software was very favorable. Many constructs that were available in the SDDL processor were not used. The requirements statements and traceability were not exercised because at the time that the design was initiated, the Software Requirements Document for the CDS had not been finished. Many of the data constructs were not used because of the specialized nature of the CDS I/O interfaces.

The implied eight dimensions given in Table 3-1 for the description of any module were never completely exercised. The one for requirements was missing because of the above stated reason. The one for system parameters was not used. This was probably because of the incomplete nature of the detailed design. The one for data specification was used in a variety of different modes. This fact leads the authors to suspect that further work needs to be done in creating a sufficient set of dimensions to describe any module.

Software design is a highly iterative process. Using SDDL as a design tool allowed the designers to easily extend, modify and update the design by making the current state of the design as embodied in SDDL very visible and easily available to all members of the design team. The ability to add new constructs (keywords) in SDDL allowed the designers to tailor the design language to the specific application.

The use of SDDL forced the designers into an organized and explicit mode of designing an associated presentation. Since the design is represented solely by the SDDL documentation of the design, it was impossible to handwave. Using SDDL, the overall aspects of the design were first captured and then refinements were added iteratively. The detailed design was captured in a document that was produced concurrently as the design evolved.

One major plus of using SDDL as a design tool as opposed to other similar tools, was the support in the design of data structures. The designers found that by first defining the interfaces and data structures, the design process was materially strengthened. Another major plus was the minimal training required before constructive work could be done by the designers on SDDL.

The single largest problem using SDDL was to produce a design document that was readable by upper level managers. However, although the document differed from that normally produced, they could read and easily comprehend the information contained in the document. Since the order of presentation in the SDDL output is completely controlled by the order of the input material, careful consideration was given to order the input in a manner that supported readability. It was also found that the casual reader needed a substantial amount of overview material. This could be done by the inclusion of TEXT blocks of information. Again, such blocks must be carefully placed within the input file. Another aspect of the stilted nature of the SDDL

output was the calling hierarchy provided by the SDDL processor. This module reference tree is complete in terms of information content. However, the casual readers found that a separate conventional hierarchy tree was required to provide this information.

APPENDIX A

SDDL DIRECTIVES

SDDL DIRECTIVES

```
: #SEQUENCE 8
: #WIDTH 130
: #MARK DATA STRUCTURE ITEMS .
: #MARK SINGULAR DATA ITEMS _
: #MARK EXTERNAL FILES @
: #MARK - $ * & ¢
: #STRING ' '
: #DEFINE MODULE REQUIREMENT ENDREQUIREMENT
: #DEFINE MODULE TASK ENDTASK
: #DEFINE MODULE GROUP ENDGROUP
: #DEFINE MODULE TABLE ENDTABLE TABLEPARAMS
: #DEFINE MODULE STACK ENDSTACK STACKPARAMS
: #DEFINE MODULE QUEUE ENDQUEUE QUEUEPARAMS
: #DEFINE MODULE FILE ENDFILE FILEPARAMS
: #DEFINE MODULE SUBROUTINE ENDSUBROUTINE EXITSUBROUTINE
: #DEFINE MODULE FUNCTION ENDFUNCTION EXITFUNCTION
: #DEFINE MODULE TEMPLATE ENDTEMPLATE
: #DEFINE MODULE PROCESSOR ENDPROCESSOR
: #DEFINE BLOCK SUBGROUP ENDSUBGROUP
: #DEFINE BLOCK REPLICATION ENDREPLICATION
: #DEFINE BLOCK SELECTION ENDSELECTION ALTERNATE
: #DEFINE BLOCK SUBFILE ENDSUBFILE , , SUBFILEPARAMS
: #DEFINE BLOCK BLOCK ENDBLOCK
: #DEFINE BLOCK PERFORM ENDPERFORM , , ALSO
: #DEFINE BLOCK ABSTRACT ENDABSTRACT , , DESIGN.DESCRIPTION
: #DEFINE BLOCK ABSTRACT , , , ALLOCATED.REQUIREMENTS
: #DEFINE BLOCK ABSTRACT , , , AFFECTED MODULES
: #DEFINE BLOCK ABSTRACT , , , OPERATING.ENVIRONMENT
: #DEFINE BLOCK ABSTRACT , , , SYSTEM.PARAMETERS
: #DEFINE BLOCK ABSTRACT , , , DATA.SPECIFICATION
: #DEFINE BLOCK ABSTRACT , , , PROCESS.SPECIFICATION
: #DEFINE BLOCK ABSTRACT , , , DATA.ORGANIZATION
: #DEFINE BLOCK ABSTRACT , , , CONTROL.FLOW
: #DEFINE BLOCK SOURCE ENDSOURCE , , VERSION
: #DEFINE BLOCK SOURCE , , , STATUS
: #DEFINE BLOCK SOURCE , , , SCOPE
: #DEFINE BLOCK SOURCE , , , PERFORMANCE.CONSTRAINT
: #DEFINE BLOCK SOURCE , , , REQUIREMENT.DESCRIPTION
: #DEFINE CALL ATTACH
: #DEFINE CALL DETACH
: #DEFINE CALL OPEN
: #DEFINE CALL CLOSE
: #DEFINE CALL CLEAR
: #DEFINE CALL READ
: #DEFINE CALL WRITE
: #DEFINE CALL GET
: #DEFINE CALL PUT
: #DEFINE CALL PUSH
: #DEFINE CALL POP
: #DEFINE CALL ENQUEUE
: #DEFINE CALL DEQUEUE
: #DEFINE CALL DELETE
```

ORIGINAL PAGE IS
OF POOR QUALITY